INTERIM
JAN-98

Semi-Annual Progress Report

January 1998

Mark R. Abbott

College of Oceanic and Atmospheric Sciences

Oregon State University

MODIS Team Member, Contract # NAS5-31360

Task Objectives

The objectives of the last six months were:

- Continue analysis of Hawaii Ocean Time-series (HOT) bio-optical mooring data, and Southern Ocean bio-optical drifter data
- Complete development of documentation of MOCEAN algorithms and software for use by MOCEAN team and GLI team
- Deploy instrumentation during JGOFS cruises in the Southern Ocean
- Participate in test cruise for Fast Repetition Rate (FRR) fluorometer
- Continue chemostat experiments on the relationship of fluorescence quantum yield to environmental factors.
- Continue to develop and expand browser-based information system for in situ bio-optical data

Work Accomplished

Analysis of Field Data

We are continuing to analyze bio-optical data collected at the Hawaii Ocean Time Series mooring as well as data from bio-optical drifters that were deployed in the Southern Ocean. As we noted in our recent quarterly report, a mesoscale feature resulted in an enormous increase in available nitrogen which apparently stimulated phytoplankton productivity. A draft manuscript has now been prepared and is being revised. A second manuscript is also in preparation that explores the vector wind fields derived from NSCAT measurements.

The HOT bio-optical mooring was recovered in December 1997. After retrieving the data, the sensor package was serviced and redeployed. We have begun preliminary analysis of these data, but we have only had the data for 3 weeks. However, all of the data were recovered, and there were no obvious anomalies. We will add second sensor package to the mooring when it is serviced next spring. In addition, Ricardo Letelier is funded as part of the SeaWiFS calibration/validation effort (through a subcontract from the University of Hawaii, Dr. John Porter), and he will be collecting bio-optical and fluorescence data as part of the HOT activity. This will provide additional in situ measurements for MODIS validation.

As noted in the previous quarterly report, we have been analyzing data from three bio-optical drifters that were deployed in the Southern Ocean in September 1996. We presented results on chlorophyll and drifter speed. For the 1998 Ocean Sciences meeting, a paper will be presented on this data set, focusing on the diel variations in fluorescence quantum yield. Briefly, there are systematic patterns in the apparent quantum yield of fluorescence (defined as the slope of the line relating fluorescence/chlorophyll and incoming solar radiation). During some periods, the slope of this line is small (indicating high productivity) whereas during other periods it is much larger (indicating low productivity). These systematic variations appear to be related to changes in the circulation of the Antarctic Polar Front which force nutrients into the upper ocean. A more complete analysis will be provided in the next Quarterly report.

MOCEAN Algorithm Documentation

As part of our joint MODIS/GLI activities, we have developed a complete set of documentation for the MODIS Ocean algorithms. This document would provide an overview of the various component algorithms as well as a brief description of the associated science. Rather than develop a paper document, Jasmine Bartlett (who was tasked with this job) developed a Web document. It can be accessed at http://ricky.oce.orst.edu/MOCP/. Figure 1 shows an overview of the algorithms. Each box is a hyperlink that allows the user to find information about each component.

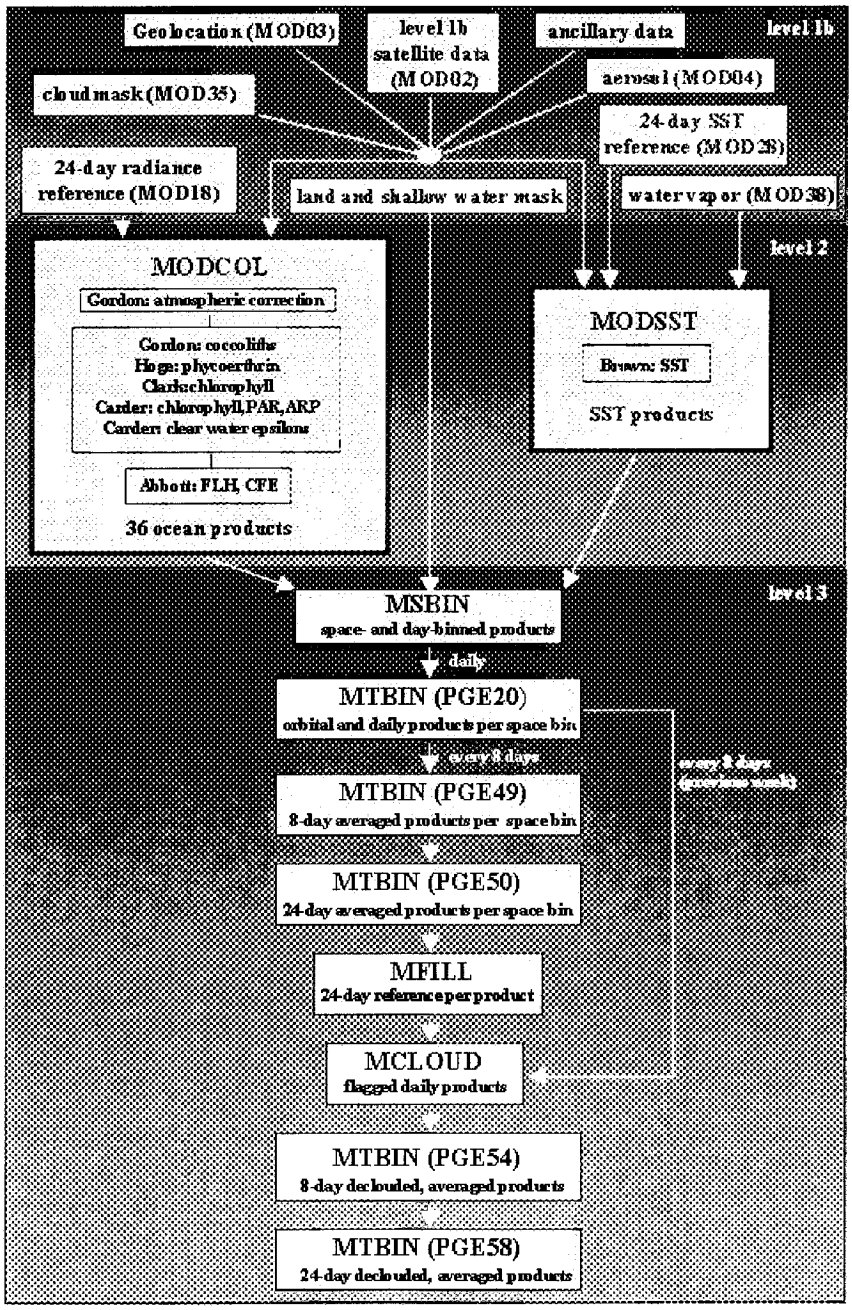


Figure 1. Diagram of linkages between components of MODIS Ocean algorithms

Rather than repeat the information here, we refer interested readers to the Web page for the complete information. The Version 2 MODIS Ocean codes will also be provided to the GLI when the NASDA researchers are ready

Instrumentation in the Southern Ocean

From October-November 1997, we participated in the first survey cruise of the Antarctic Polar Front as part of JGOFS. We deployed 12 moorings, each equipped with a current meter and an irradiance sensor. Six of the moorings also had a conductivity/temperature sensor. These moorings were deployed in a grid with spacing of approximately 30 km between each mooring. We also deployed 10 bio-optical drifters and 10 conventional drifters. Unfortunately, two of the bio-optical drifters failed on deployment, but the vendor has provided us with free replacements for future experiments.

The most recent set of drifter tracks is shown in Figure 2. Note that the main cluster of drifters follows the PF, showing strong divergence near 160°W, associated with the Pacific Antarctic Rise. We expect that primary productivity will be stimulated by the upwelling forced by this divergence. Note also that several drifters are "kicked" out of the PF to the north. This may be an important mechanism for meridional exchange of heat and nutrients.

The Tethered Spectral Radiometer Buoy II was deployed at several stations during the cruise. Chlorophyll values were generally low, given the deep mixing present in the PF during early spring. The TSRB II values agreed quite well with chlorophyll extractions made from near-surface water samples. Sun-stimulated fluorescence was also measured, and these data are now being analyzed.

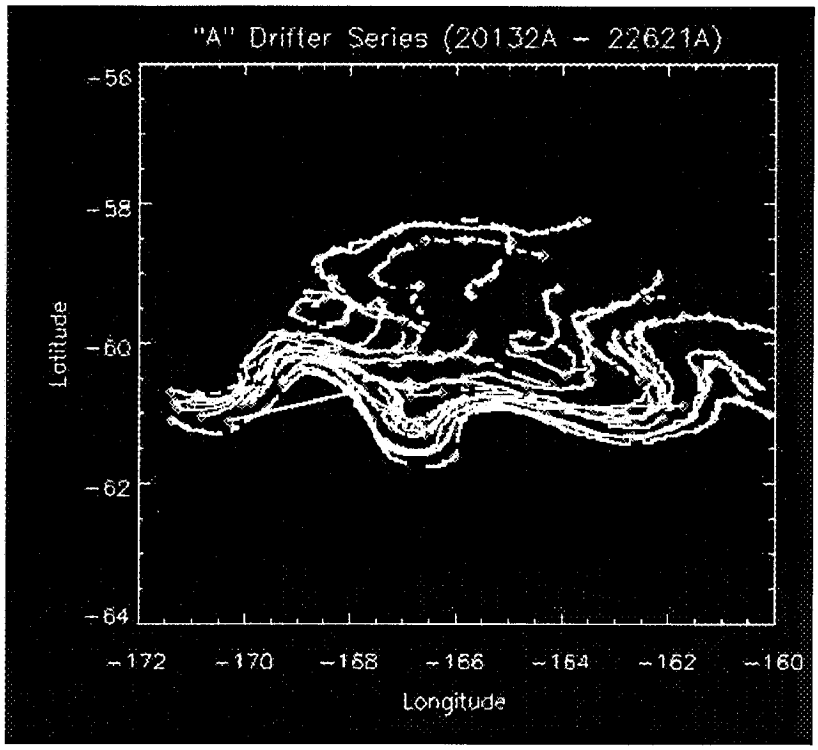


Figure 2. Tracks from drifters deployed in November 1997, as of January 1998.

Figure 3 shows the positions of the moorings and the drifter tracks from the first few days of the deployments. This information is overlain on a map of SST derived from SeaSoar measurements. Note the strong meandering of the flow field, which is associated with upwelling and downwelling. Initially, the drifters pass through a downwelling region which is associated with strong convergence of the drifter tracks. This is followed by upwelling in a divergence region in the meander. This area of upwelling had high chlorophyll levels. The iceberg shown in Figure 3 may have damaged one of our moorings, but we will not know for sure until the recovery cruise in March 1998.

Fast Repetition Rate Fluorometry

The first test with the Fast Repetition Rate (FRR) fluorometer was a qualified success. Although the data were reasonable, there were several technical problems with the operation of the FRR fluorometer. Most notably, the underwater connectors and some elements of the software were poorly designed. The fluorometer was shipped back to Chelsea Instruments in the United Kingdom, so it was not available for the first Polar Front survey cruise.

The FRR fluorometer was returned in December and is now on the R/V Roger Revelle as part of the second JGOFS PF survey. All of the connectors were replaced, and the software has been rewritten. Initial reports are that it is working perfectly, and pigment samples are being collected for HPLC analysis as well as fluorometric analysis.

Chemostat Experiments

We have completed a major redesign of the chemostat that we are borrowing from Dr. Dale Kiefer. This included:

- Redesign of the electronics backplane for the chemostat so that components can be installed and serviced more easily
- Improved electrical fusing and cooling capabilities of the chemostat
- Tested and improved stability of optical components
- Rewrote software to be Win32 compatible to take advantage of real-time capabilities of Windows NT for data acquisition and system control
- Acquired quantum probe for PAR measurements and pH/temperature probe to monitor cultures
- Developed stirring system for phytoplankton cultures
- Developed approach for autoclaving culture media so as to maintain axenic cultures

We have now begun initial tests with the rebuilt chemostat.

EOSDIS Plans

Attached in the Appendix are two papers that have been submitted to the Association of Computing Machinery (ACM) workshop on Java and High Performance Computing. These papers document the status of our advanced browser activity.

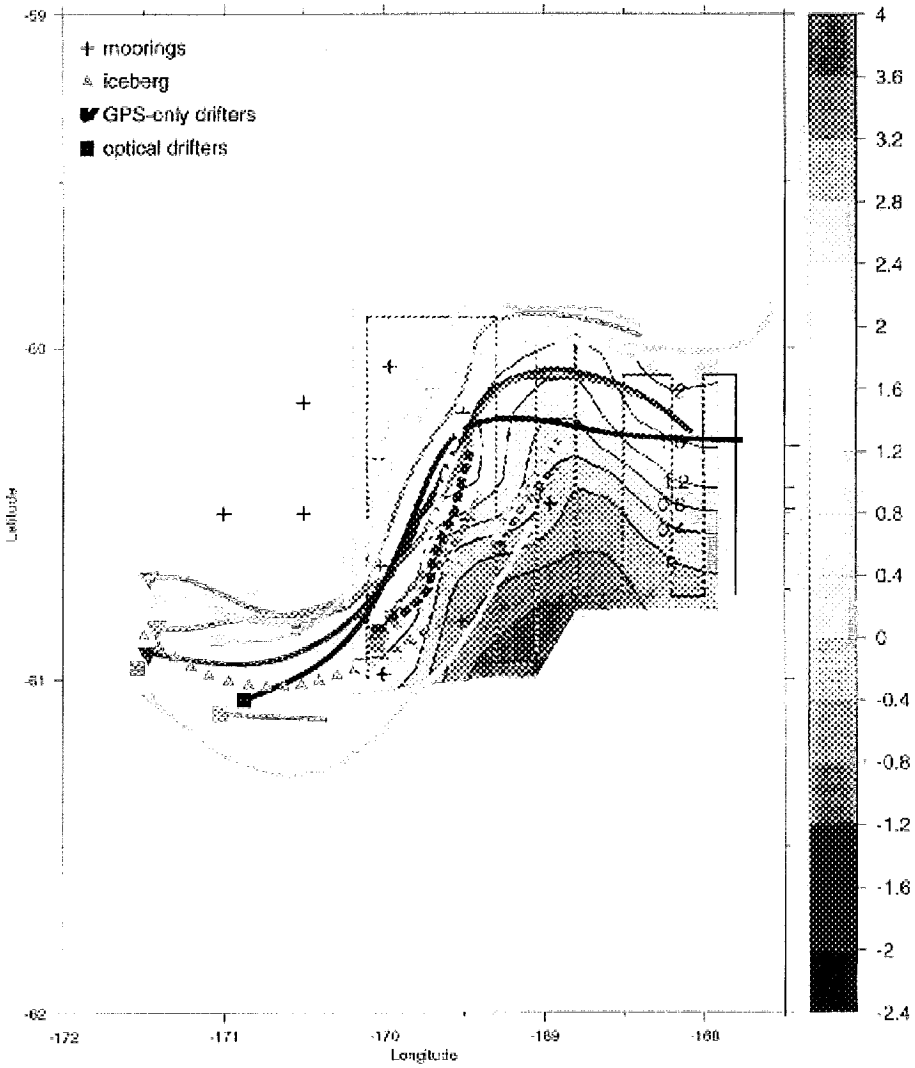
Along with the MODIS Web documentation, we are revising our archive of satellite and in situ data sets. With the release of SeaWiFS data, we are now assembling a data base of both SST and chlorophyll for the Southern Ocean region. This will also include our bio-optical drifter and mooring data.

Anticipated Future Actions

- Retrieve and redeploy bio-optical mooring in Hawaii and continue analysis of bio-optical data
- Analyze data from bio-optical moorings and drifters, TSRB II, and FRR in the Antarctic Polar Frontal Zone
- Continue chemostat experiments on the relationship of fluorescence quantum yield to environmental factors. Establish relationship between fluorescence quantum yield and photosynthetic parameters.
- Continue to develop and expand browser-based information system for in situ bio-optical data.

Sea Surface Temperature (°C) SOJGOFS Survey 1, 15-17 Nov 1997

Drifters from 8-18 November 1997



Preliminary

Oregon State University SeaSoar Group
Oregon State University Mooring Group

Problems and Solutions

The most significant concern remains the apparent inability of EOSDIS to deliver data products at launch. The present approach to cost-savings is based on scaling back hardware acquisitions, which has been shown to be a small fraction of the overall EOSDIS budget. Thus the approach mandated by NASA Headquarters will likely not save money while at the same time causing deep frustration in both the EOS and general Earth science communities. We are concerned that insufficient data will be delivered for algorithm validation as well as analysis in support of future EOS sensor designs.

Appendix

Manuscripts submitted to *Association of Computing Machinery Workshop on Java and High Performance Computing.*

DOVE: Distributed Objects based scientificVisualization Environment

Mark Abbott, Lalit Kumar Jain

College of Oceanic and Atmospheric Sciences
Oregon State University
Corvallis, OR 97331
{mark, jain}@oce.orst.edu

Abstract

This paper describes the design and performance of a distributed, multi-tier architecture for scientific data visualization. A novel aspect of this framework is its integration of Java IDL, the CORBA distributed object computing middleware with JavaBeans, the Java Component model to provide a flexible, interactive framework for distributed, high-performance scientific data visualization. CORBA server objects running in a distributed collaborative environment provide data acquisition and perform data-intensive computations. Clients as Java Bean components use these server objects for data retrieval and provide an environment for visualization. The server objects use JDBC, the Java application programming interface to SQL databases, to retrieve data from the database. We discuss the system framework and its components and describe an example application and its performance.

Keywords: Distributed Object Computing, Component model, Multi-tier architecture, Scientific data visualization, CORBA, Java IDL, JavaBeans, JDBC, SQL database.

1 Introduction

Two key requirements of an interactive visualization system for scientific data exploration are flexibility and performance. The purpose of a data exploration system is to enable users to uncover and extract relationships hidden in large data sets [2]. This usually requires flexible data manipulation mechanisms which provide a complete coverage of operations needed by the users. The tools available to users should enable them to form ad hoc groupings of data so as to facilitate understanding of the relationships latent within the data. Performance is the other key requirement. Scientific data visualization systems usually require bulk data transfer and involve intensive computation. Designing a system which meets the response time requirements of an efficient, high-performance system is a challenge. In such settings, recent technological advances in areas of component-based software development and distributed object computing have proved highly beneficial. Component-based software development is a promising technology which is gaining rapid popularity because it naturally extends the object-oriented paradigm emphasizing on modularity and reusability. Components are self-contained reusable elements of software that can be controlled and assembled dynamically to form applications [2]. Components can be customized and interactions between them defined at design or run time providing a flexible framework for application development. On the other hand, the distributed computing paradigm brings in the idea of distributing the computation by harnessing geographically-remote resources. Blending the flexibility of component-based software development with the power of distributed object computing results in a flexible, efficient distributed computing environment for scientific visualization.

Java [1] is a simple, object-oriented, architecture-neutral, portable and multithreaded programming language. It is almost the ideal language for writing portable client/server objects. But it needs an intergalactic distributed

object infrastructure to complement its capabilities for high-performance, distributed object computing. This is where CORBA comes into picture. CORBA is an emerging standard for distributed object computing sponsored by the OMG [4]. CORBA provides a flexible high-level distributed object computing middleware. Pools of server objects can communicate using the CORBA ORB. These objects can run on multiple servers to provide load-balancing for incoming client requests. CORBA's distributed object infrastructure makes it easier to write robust networked objects and coupled with Java's built-in multithreading helps achieve high performance.

The JavaBeans architecture [5] brings the much needed component development model to Java. JavaBeans is a platform-neutral architecture which extends Java's "Write Once, Run Everywhere" capability to component development. The technology enables developers to create reusable software components which can be visually manipulated and dynamically assembled in a wide variety of application builder tools to form working applications.

2 Framework for Distributed, Three-tier Scientific Visualization:

2.1 Example Data Domain

This work is being done as part of the Earth Observing System (EOS) project [6] at the College of Oceanic and Atmospheric Sciences (COAS), Oregon State University. EOS is part of NASA's effort to study global climate change. The project represents a continuous effort by scientists in different parts of the country to carry out studies on a variety of Earth and Space Science data sets collected over periods of time, analyze these data sets and thereby infer changes in global climate. The data sets represent different kinds of data. Some example data include numerical data from the ocean recorded by sensor instruments called drifters that float in the ocean and record various parameters like sea surface temperature, chlorophyll content etc., satellite imagery taken by polar-orbiting satellites that cover the world ocean. Three dimensional data sets include parameters of the ocean like salinity, pressure and temperature at various depths in the ocean recorded by the National Oceanic and Atmospheric Administration (NOAA).

2.2 Architectural Design

We have a three-tier framework with the client visualization components in the first tier, the application servers in the middle tier and the database in the third tier. The application servers run as distributed objects on multiple servers in a heterogeneous environment and embody most of the logic related to data retrieval and computation. Thus, the client or the front-end becomes very thin, making it suitable to run on low-end machines such as lightweight PCs and the JavaStation. The client can run either as an application or as an applet.

The innovative part of our framework is way in which the client components interact with each other and with the server objects and how the server objects distribute the process of data retrieval and computation among themselves to provide a flexible, high-performance environment. The application servers run as distributed CORBA objects. Each type of server provides uniform access to a particular type of data through the use of a database handler which encapsulates database specific querying logic. In addition, it can also perform computation intensive tasks efficiently by distributing the task among a set of available computation servers running on a High Performance Cluster (HPC). Several instances of each type of server may be running on multiple heterogeneous machines.

The front-end consists of a set of client components, each of which acts as a visualization tool for a particular type of data, and a set of filter components which are used by client components for various types of data manipulation. The client components communicate with the server objects for data retrieval. These client components implement interfaces allowing them to interact with each other to provide a basis for data exploration. Such interactions may be direct communication between client components involving transfer of data or may be indirect through a filter component wherein data is 'filtered' before being passed on to the other component.

3 DOVE System Components

The *Dispenser* :

The Application Server tier (middle tier) consists of a set of CORBA objects running in a distributed heterogeneous environment. These server objects being CORBA objects can be written in any of several languages having CORBA support and may use any of the available ORBs. The client components in the front-end communicate with these server objects for data retrieval. The *Dispenser* is a CORBA object manages these server objects and acts as a registry which the clients can use for accessing a reference to the server objects. It implements the following interfaces:

Register Interface: This interface is used by the server objects to register themselves with the *Dispenser*. A server object registers its unique Interoperable Object Reference (IOR) with the Dispenser. Multiple instances of the same server type may register with the Dispenser.

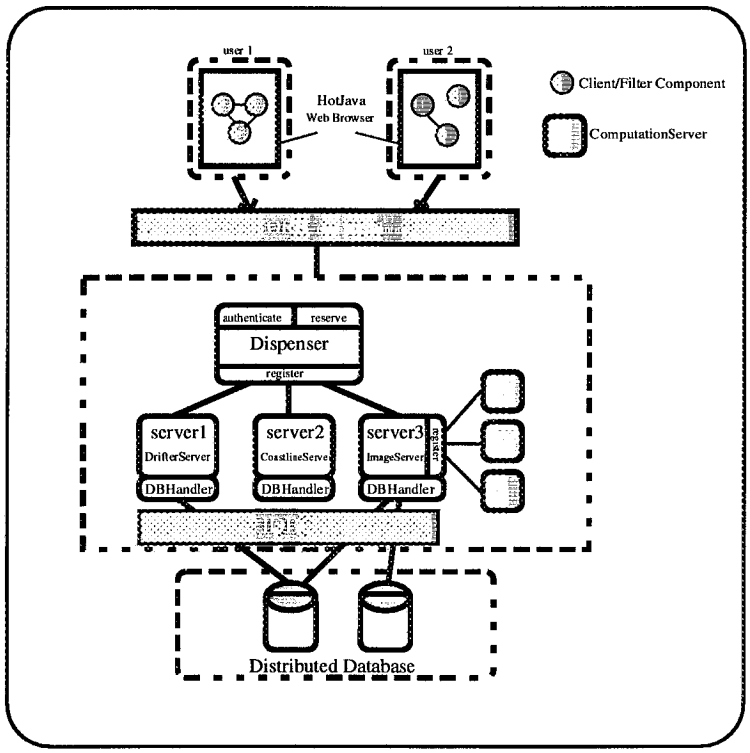


Figure 1: DOVE System Architecture

Authentication Interface: Clients need to be authenticated before being allowed access to server objects because the server objects provide direct access to the database.This interface lets a client authenticate itself with the Dispenser allowing it to request a reference to a server object. The Dispenser maintains a registry of authorized users.

Scheduler Interface: The scheduler interface is used by clients to get CORBA references to server objects. The scheduler incorporates load balancing logic to select a server object with minimum load in case multiple instances of that server object are registered.

Application Server Objects* and *Server Proxies : An *Application Server Object* is a CORBA object. Each *Server Object* provides uniform access to a portion of the database representing a dataset. A *Server Proxy*, associated with every *Server Object*, encapsulates CORBA-specific code required for communicating with the *Server Objects*. A client component creates an instance of a *Server Proxy* to communicate with a *Server Object*. Upon creation, a *Server Proxy* gets a reference to its *Server Object* which it will use for further communication with the *Server Object*.

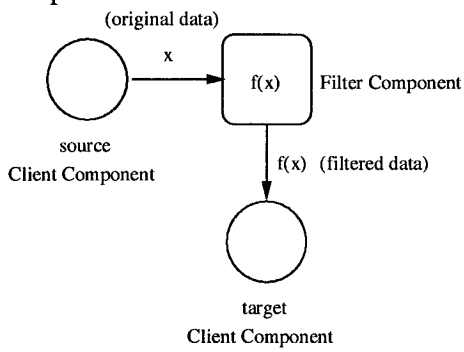
Database Handlers : Every *Server Object* uses a *Database Handler* for communicating with the database. We use JDBC [9] for connecting to the database and querying it. A *Database Handler* converts Client queries into database specific queries and returns the results back to the *Server Object*.

Computation Server Objects : A *Computation Server Object* assists an *Application Server Object* in performing computation-intensive tasks. A *Server Object* can distribute its task among multiple *Computation Server Objects* depending upon the computational complexity of of the task and the number of *Computation Server Objects* available at runtime. Java's multithreading capability help a *Server Objects* to invoke multiple *Computation Server Objects* simultaneously by invoking each of them in a separate thread.

Client Workspace : The *Client Workspace* is basically a Java Beans container providing a context in which the *Client Components* can interact with each other. In our implementation, the *Client Workspace* is a customized version of the the Java Bean Box container that is part of the Beans Development Kit (BDK). We have customized the BeanBox so that it runs as an applet in the browser. The *Client Workspace* supports remote serialization allowing a client to save his workspace into the remote database for later use.

Client Components* and *Filter Components : A *Client Component* is a Java Bean Component which runs in the *Client Workspace*. It interacts with the user and can be linked with other components in the *Client Workspace*. It communicates with one or more *Application Server Objects* to retrieve its data. A *Filter Component* is also a Java Bean Component which runs in the *Client Workspace*. It is meant to be used in conjunction with *Client Components* to produce filtered versions of datasets. It is associated with a source *Client Component* and a target *Client Component*. It takes data from the source, filters/manipulates the data according to some algorithm implemented by it and returns the data to the target (figure 2). A source component can itself be the target of a *Filter Component*. The framework allows a user to define his own filter and use it in the workspace.

figure 2: Component Interaction involved in Data Filtration



4 Technical Issues

Designing and developing a distributed framework is a difficult task. There are many technical challenges related to functionality, availability, reliability, security and performance which needs to be addressed. In this section, we briefly describe some of the design issues related to the proposed framework and show how they are addressed.

4.1 High Availability

The front-end Client Components rely on the Application Server Objects for data retrieval. Non-availability of Server Objects would result in inaccessibility of the database. One technique for improving Server availability is replication of Server Objects. In our system, the Dispenser allows multiple Servers Objects of the same type to register with it.

4.2 Reliability

Distributed application often require substantial effort to achieve levels of reliability equivalent to those expected from stand-alone applications. Detecting failures in distributed application is quite complicated. In our system, an Application Server Object might become unreachable after it registers with the Dispenser. However, the Dispenser continues to retain the Server's reference assuming its availability. So, detecting failures of Server Objects is needed. In our system, every Application Server Object implements a method called isAvailable() which returns a boolean true. When a client request for a Server reference comes in, the Dispenser invokes the Application Server Object's isAvailable() to determine its availability (the Server Object would return true if available, otherwise an exception would be thrown by the ORB which would be caught by the Dispenser). If the Server Object is not available, the Dispenser removes the Server from its registry and tries for other Server Objects of the same type.

The other problem is that the Dispenser itself may crash. In this case, the problem is that the Application Server Objects which have registered with the Dispenser have no idea about it. So, they would not try to register again with the Dispenser. To overcome this problem, the Dispenser serializes its registry (using Java's object serialization) into a file every time an Application Server Object registers with it. Whenever the Dispenser starts, it checks for the registry file and reloads any serialized data available in it.

4.3 Load Balancing

Load balancing techniques try to dynamically balance the system workload among various participating servers. The DOVE system has minimal load balancing capability which involves routing a client's request to a server which is currently not busy. The Dispenser marks a server as busy when it is allotted to a client in response to its request for server reservation. The server is marked as not busy when the client releases the server by invoking a method on the Dispenser for releasing the server.

4.4 Interoperability

Every ORB provides its own implementation of the CORBA COS (Common Object Services) naming service which can be used to store and retrieve object references. However, these nameservice implementations are not interoperable; that is, objects written using one ORB cannot register with or look into the nameservice of another ORB implementation. For example, if an object A which uses the Java IDL ORB [10] needs to communicate with an object B which uses the VisiBroker ORB [7] and B is registered with the VisiBroker's directory service *osagent*, the Java IDL object A cannot access the *osagent* to get a reference to B. Inclusion of the Dispenser object in our design is mainly to resolve this issue.The Dispenser object implements a register interface which acts as a common nameservice allowing objects written using different ORBs to register with it so that they are accessible by any client in the framework.

4.5 Security

The Application Server Objects provide access to the database. Thus clients need to be authenticated before providing Server Object references to them. In our implementation, the Dispenser object authenticates a client before giving an object reference to it. The Application Server Object understands that only authenticated clients have been allowed to access its services by assuming that a client cannot guess any Application Server Object's IOR. However this assumption may not be true, requiring every Application Server Object, in addition, to provide to its own authentication interface.

5 Example Application and Performance Measurements

In this section, we discuss about the working of a particular application we have developed using the proposed framework and briefly look at the performance measurements. The application consists of the following Application Server Objects and Client Components:

Application Server Objects:

1. *DrifterServer* : This provides access to the drifter data such as the position of a drifter and values measured by it.

2. *CoastlineServer*: This provides access to the coastline data corresponding to a given region.

3. *ImageServer*: This provides access to the satellite imagery taken by satellites.

4. *ImageProcessorServers*: These are the computation servers used by the ImageServers for data intensive computations which includes decompressing a GIF image, manipulating its pixel values and/or color map and compressing it back into the GIF format. Such computation is required to provide an animation of the satellite images with the drifter track overlaid on it based on region and time. It requires decompressing each image, manipulating its pixels and color map to draw the track on it, and compressing it back into the GIF format.

The Server Proxies associated with these Application Server Objects are named *DrifterProxy*, *CoastlineProxy* and *ImageProxy*.

Client Components:

1. *TrackWizard*: This component uses the DrifterProxy and the CoastlineProxy for communication with the DrifterServer and the CoastlineServer respectively, and provides a two dimensional plot of the track taken by a drifter (or a set of drifters) in the ocean along with the coastline corresponding to that region. Java's built-in multithreading helps in retrieving both the drifter data and the coastline data in parallel by communicating with the DrifterServer and CoastlineServer simultaneously in different threads.

2. *ImageWizard*: This component uses the ImageProxy to communicate with the ImageServer and provide display and animation of satellite imagery.

3. *AlgoBean*: This is a Filter Component which modifies the drifter data according to an algorithm. The TrackWizard could be its source components. The target components could be the ImageWizard or the TrackWizard itself.

A very good example which demonstrates the flexibility of Java Beans in this application is addressing a user's need for a mechanism by which he could overlay the drifter track on the images based on region and time and view an animation. In the absence of the component model, a naive solution would be to develop a new component for the specific purpose which would retrieve both the drifter data and image data and perform the computation to display the animation. The component model enables us to design the components so that the user can just drag the drifter track from the first component and drop it on the second component, and perform the necessary computation to provide the animation.

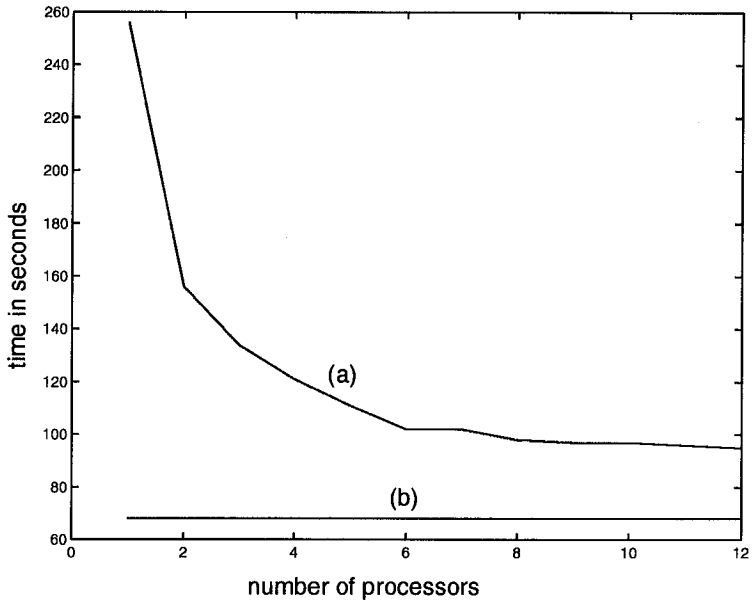
Performance Measurements

We conducted experiments to calculate the speedup achieved by distributed computing. The experiments we conducted to test the performance of the ImageServer object which basically performs two tasks, viz., image retrieval and image processing. Image processing is a computation-intensive task involving image decompression and compression and hence a suitable candidate for distributed computing. The ImageServer performs the image retrieval on its own, but distributes the image processing task among a set of available

ImageProcessorServers. Each image can be processed independent of the other images retrieved in the same query. So, the processing of images can be performed in parallel to image retrieval. The ImageServer divides the total number of images to be processed (say n) among the number of dynamically available ImageProcessorServers (say k). After retrieving every bunch of n/k images, they are handed to an ImageProcessorServer for processing. The important point to note here is that the image retrieval time remains constant even though the image processing time varies according to the number of available ImageProcessorServes. However, in the experiments conducted we measured the total time which include image retrieval time and image processing time.

The experiments were conducted on a network of computers consists of Sun UltraSparcs. The HotJava browser was used to run the clients.

Figure 3 shows the performance curve for the test runs.



(a) total time (image retrieval time + image processing time)
(b) image retrieval time

Figure 3: Performance curve: number of processors versus time.

6 Future Work

Java has evolved has a set of APIs for specific purposes. Developing a scientific visualization environment requires dealing with varied fields such as networking, databases, graphics and animation. We have exploited Java's capabilities in all these areas. However the core Java APIs lack a powerful set of classes for image processing, though our system provides visualization capabilities such as two-dimensional plots, image processing, animation, colormap customization, etc. We are awaiting the release of Java Media APIs which will enable us to view our data in different formats such as digital video. We are also looking forward to using the Java 3D API which will provide us with higher level constructs for visualization of 3D data sets [8]. Support for running Java in embedded devices (such as ocean drifters which forms one of the main sources of our data) will add further capabilities such as real-time visualization to the DOVE system.

7 Conclusion

In this paper, we have described a flexible and efficient distributed computing environment for scientific data

visualization. CORBA server objects provide access to database and perform data intensive computation. Clients, implemented as Java Bean components use these server objects for data retrieval and provide data visualization. Our earlier framework was based on Java applets and Java RMI which was not flexible because of limited interaction possible between applets. The use of the Java Beans component model and the CORBA middleware has made our architecture very flexible and powerful. But currently there is no Java Beans application builder tool which can run in the browser. We customized the Java BeanBox container (which is part of the BDK 1.0) so that it runs as an applet allowing components to be assembled on the web.

References

[1] James Gosling, Henry McGilton, "The Java Language Environment" , A White Paper, May 1996.

[2] Jade Goldstein, Steven F. Roth, "A Framework for Knowledge-Based Interactive Data Exploration", Carnegie Mellon University.

[3] Robert Englander, "Developing JAVA Beans", The Java Series, O'Reilly, June 1997.

[4] Object Management Group, The Common Object Request Broker: Architecture and Specification, 2.0, July 1995.

[5] Sun Microsystems, JavaBeans specification via
http ://java.sun.com/beans/docs/spec.html

[6] Earth Observing System project, http://eos.nasa.gov

[7] VisiBroker for Java: Programmer's guide, Visigenic Software
http://www.visigenic.com/techpubs/htmlhelp/vbj30/pg/noframes/vbjpgmrt.htm

[8] Sun Microsystems, The Java 3D API White Paper, via
http://www.javasoft.com/marketing/collateral/3d_api.html

[9] Sun Microsystems, The JDBC Database Access API specification 1.10 via, http://java.sun.com/products/jdbc/

[10] Sun Microsystems, The Java IDL via
http://java.sun.com/products/jdk/1.2/docs/guide/idl/index.html

MIDAS – A Mobile Intelligent Data Acquisition System

Vic Anantha, Mark Abbott

College of Oceanic and Atmospheric Sciences
Oregon State University
Corvallis, OR-97331
{anantha, mark} @oce.orst.edu

Abstract

MIDAS (Mobile Intelligent Data Acquisition System) is a Java based framework used to obtain data from remote sensors. One of the major problems with remote data acquisition systems has been the lack of interactive communication between the user and remote sensor. Advancement of embedded system technology has motivated us to build an experimental system having two main features. The first, being intelligence on the sensor which can detect a change in conditions and accordingly modify the sensing characteristics. The second feature is the capability of a remote user to upload code and start remote execution of the code on the sensors. Java allows us unify the data acquisition components with the data visualization components using simple and standard programming methodologies. At the time of writing the paper, the software framework for MIDAS has been developed. Some of the connectivity issues are presently being addressed.

Introduction

Remote data acquisition for oceanographic purposes can roughly be divided into two categories, drifters and moorings. The more sophisticated instruments, which involve constant human supervision, are not mentioned here. Moorings are fixed and data at only one location is acquired over a period of time. A system of moorings placed at pre-defined locations can give a global view of the region. Drifters on the other hand are allowed to drift with the ocean currents, recording data periodically. Typically, data retrieval from the moorings has involved copying data from the on board non-volatile memory after retrieving the instrument.

Satellite communications are becoming more common, but this has involved only a one way communication from the sensor to the host computer. ARGOS (Advanced Research and Global Observation) [1] is a satellite-based location and data collection system dedicated to monitoring the environment. ARGOS is currently being used at the College of Oceanic and Atmospheric Sciences for collecting drifter data. ARGOS is capable of locating any mobile device carrying a suitable transmitter anywhere in the world to within 150 meters. The ARGOS instruments are flown on board the National Oceanic and Atmospheric Administration (NOAA) [2] Polar Orbiting Environmental Satellites (POES). At least two satellites are operational at any time. Users can subscribe to the ARGOS service and incorporate the ARGOS transmitters into their remote sensing units. The satellites receive the messages from the users' transmitters and relay them to the ARGOS ground stations. The data is then transmitted from the ground station to the user at regular intervals. The satellites can make fourteen passes around the Earth every day. This model is not interactive and does not allow the user to send commands to the remote sensing unit. Commercially available CDPD (Cellular Digital Packet Data) networks and radio LAN adapters seem to be a viable option for establishing two way communication between the sensors and the host computer.

The smart sensor in MIDAS requires an embedded system running a standard operating system. It should be able to support the sensing devices and the radio LAN adapters. It should be

lightweight and should have low power consumption characteristics. These components essentially make up a mobile computer. Mobile computing [3] is characterized by resource constraints, variable networking connectivity, and a finite energy source. Mobile clients have to make a compromise between autonomy and collaborative control. MIDAS exhibits some problems which are characteristic of mobile computers. Due to the larger latencies and unreliability inherent in mobile computing we are faced with the classic command and control problem. The knowledge of the system might be obsolete by the time a command reaches it. So, a predicate, which must be true when the command reaches the sensor, should also be sent along with the command. This is similar to the hints used in distributed systems. [4]

The design of the system was biased towards the use of Java because we required code mobility and platform independence. In this scenario, an embedded system running JavaOS [5] or a system with a JavaChip [6] would be the ideal platform. In the absence of commercially available systems using this technology at the time of writing this paper, we used a system running WindowsCE. [7]

Motivation

The main reasons for building a self-correcting system, which is also capable of receiving commands from a user, are:

1. Remote data acquisition systems are tuned to sample data at a particular rate. This rate is based on expected behavior determined by simulation and modeling. These models are bound by the fact that the behavior being modeled keeps changing. The most effective system for monitoring of such features will be a dynamic model that is continuously updated by the assimilation of data [8]. In such a scenario it would be very useful to have a smart program running on the remote system which can dynamically change the sampling characteristics based on the feedback received from the sensors.
2. Power consumption is a major concern in these devices. If the rate of change of the measured property is fairly constant, the sampling rate can be reduced, lengthening the battery life. The data can be trusted only if the available power is above a certain threshold value.
3. The investigator must have control over the data acquisition system. She should be able to send commands to the system. These commands could be either to change the sensing behavior, or to determine what data has to be transmitted. The software residing on the smart sensor should not be too complicated because excessive computation on the sensor site would increase power consumption. Hence our goal is to identify some core functionality which can be performed by the sensor without excessive demands on power. Allowing investigators to interactively send commands to the sensor, when they identify interesting behavior provides additional flexibility.
4. Sometimes, the actual properties being measured are not of direct interest. So a derived value, which is of primary interest could be calculated on the fly and then transmitted. This could potentially save transmission time. For example, certain experiments involve the measurement of different wavelengths of light. But the actual values of interest are the ratios of different wavelengths.
5. Currently, systems like ARGOS do not use a robust data transmission protocol. A raw stream of bits is transmitted without any attempt at error correction and retransmission. Only error detection is possible. As a result data obtained is frequently unreliable. A smart sensor using a standard communication protocol like TCP/IP can assure a higher quality of data.

System Architecture

Distributed objects are used to design the mobile computing system. [9] The system consists of a number of mobile clients interacting with a server, which has a wired connection to the network. The connectivity between the client and the server is wireless. We are currently investigating two possibilities for the wireless connection. Using a commercial CDPD network would mean that the service provider could handle problems like network handoff. But then, the network coverage offered is limited. Using a custom radio LAN configuration gives us higher network coverage but then MIDAS has to handle the problem of network handoff.

MIDAS uses a three tiered architecture comprising of the following components:
Third Tier: There are two different third tier components viz. the mobile client and the database. The mobile client comprises an object, which performs the following functions.

1. It reads data from the sensor at periodic intervals and stores it onto a local cache. This interval can change dynamically depending on the quality of data obtained. Certain other device specific parameters can also be changed dynamically. The change can be self initiated or can be initiated by the user by sending a command.
2. Commands from the middle tier are accompanied by a predicate, which must be true, when the command has been received. The third tier component listens for commands from the middle tier and evaluates the accompanying predicate. If this predicate is true, it performs the action specified in the command. The commands can be to transmit data or to change some device specific characteristics. These commands are implemented as Java threads.

The mobile client object uses sockets to listen to specific instructions from the middle tier. This object is capable of uploading Java class files from the middle tier and executing them. These uploadable Java class files conform to a particular pattern. All of them implement a predicate method, which indicates if the conditions observed by the user are still true. They are also implemented as threads so that they can be started and stopped independently. A vector containing all the threads residing on the system and their status (running or stopped) is stored on this object. There are different types of instructions that the mobile client object is capable of executing:

1. Start execution of a thread.
2. Stop Execution of a thread.
3. Upload the class file from the middle tier.

Depending on the success or failure of a particular instruction a specific result code is returned to the middle tier.

The Database is also a third tier component. The data, which is retrieved from the sensor, is stored in it. It is essentially a SQL compliant database, which can support queries from Java objects. JDBC provides the functionality to query the database. If the native database drivers use the ODBC paradigm, then a JDBC-ODBC bridge is required to transform the JDBC calls to conform to ODBC standards.

Second Tier: The second tier exists on a static host and comprises of two objects.

1. Gateway object: this acts as a gateway for the mobile clients. It communicates with the third tier using the protocol described in the previous section. It periodically retrieves the data from the third tier and stores it into a database. It also packages the commands to change device characteristics into an object and uploads it to be executed on the third tier.

2. Database-query object: this takes query parameters from the user, retrieves the data from the database and sends it back to the first tier component.

First Tier: The first tier consists of two Java applets embedded in an HTML page:

1. Visualization object: It is used to build the database query of the data the user is interested in and create visualizations of the retrieved data.
2. Operational command object: Only authorized users can call this code. This client control is used to create the operational commands, which can be uploaded to the sensors. The user defines some conditions, which are then transferred to the middle tier Gateway object, which translates it. The Gateway object then constructs the class file conforming to the pre-defined pattern and uploads it to the third tier object.

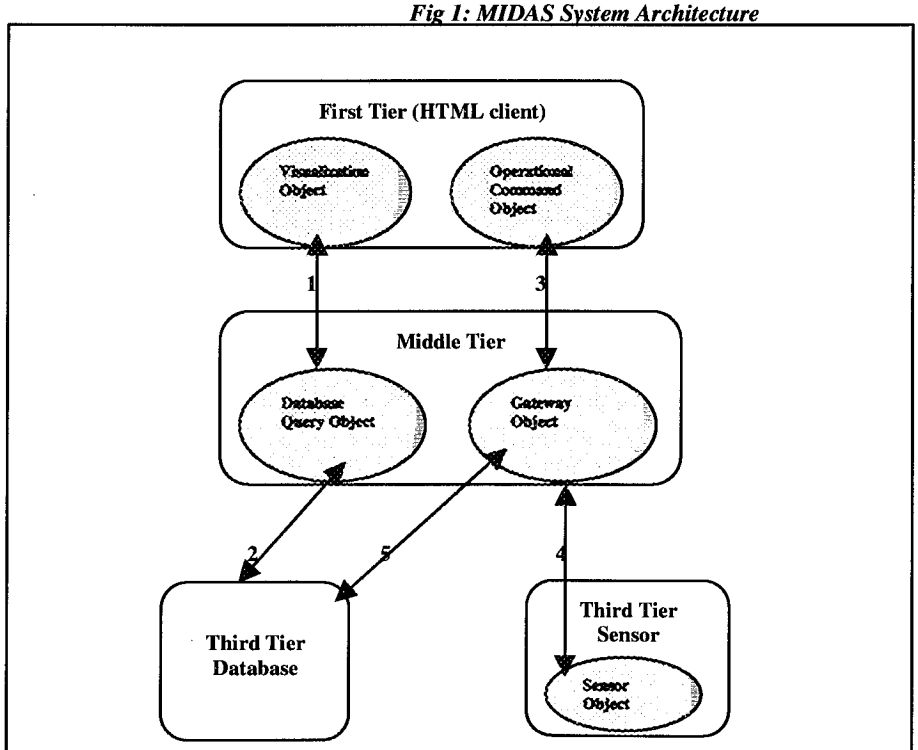
The first tier has been designed to operate on thin clients, so most of the functionality is performed on the middle tier. The first tier is essentially used for display of results and user input.

In a typical scenario, the third tier Sensor object would be running the following threads.

- A thread to read data from the serial port at regular intervals.
- A thread to read data from the non-volatile memory and send back to the middle tier.
- A thread which computes the rate of change of the properties being measured and makes some intelligent decisions about changing the sampling rate and other device characteristics.

If the investigator discovers some phenomenon which is not being handled by the above thread, she can invoke the first tier client on a web page and specify the conditions to check for and the resultant action to perform. These specifications are sent to the middle tier. The condition to be checked for is converted into the predicate method of the class. The class file meeting these specifications is built and then transferred to the third tier. The third tier builds a class from these bytes, creates an instance of the class, makes appropriate entries into a vector and then invokes the start() method of the downloaded object.

Fig 1: MIDAS System Architecture



1 – User sends query information to Database Query object and gets back the query results using RMI.
2 – Database Query Object makes query to the Database and gets back the results using JDBC calls
3 – Researcher specifies the commands to be executed on the sensor, sends the specification to the Gateway object and gets back the result code using RMI.
4 – Gateway object packages the commands and sends them on to the Sensor object. It gets back the result codes and the data from the sensors. This communication is done using sockets.
5 – Gateway object updates the database with data received from the Sensor object using JDBC calls.

Applicability of Java

This section discusses the rationale behind using Java as the implementation tool. The commands, which are uploaded from the middle tier, are essentially mobile network agents. [10] Hence there is an inherent advantage of using an interpreted language. The class files which are built on the middle tier can be transferred to the third tier mobile client and interpreted without having to compile the code for the new platform.

Java is an object-oriented language, which has a cleaner programming model than C++. The support for network programming is extensive. The sockets API is very easy to use from the programmers' point of view when compared to the sockets API in C. The ClassLoader class can be subclassed so that classes can be downloaded from the network. This simplifies the process of uploading the commands from the middle tier and executing them.

Support for multithreading allows different threads of execution to occur simultaneously. This model allows the mobile client object to start the execution of a command thread and then get back to listening for more instructions from the middle tier. Although other languages offer support for multithreading, the ability to use it very easily encourages programmers to make their applications multithreaded.

Java also recognizes that in certain cases it would be more efficient to use C code and the provisions to use native code allows programmers to use C programs for some device controllers and other low level code. Thus there are some clear advantages to using Java in this environment.

Implementation details

In this section, we talk about the implementation details and offer some programming insights. The low-level code used for interfacing with the sensors was written in C. It uses the Win32 API native to the Windows world. This was accessed from the Java code using the native interface. Windows CE is 32-bit operating system, which runs entirely from ROM. It supports many different types of device interfaces.

The platform supports the NDIS 5.0 specification for the network interface. Many companies have released the wireless LAN adapters confirming to the PC type II specification. But power consumption of these devices is a matter of concern. The connectivity issues are yet to be worked out. Many of the current drivers for the PC cards are still in the beta-test stage. One of the options is to use a wireless devices confirming to the WLIF (Wireless LAN interoperability forum) [11] standard. They use Frequency Hopping Spread Spectrum technology for the wireless medium. It operates in the unlicensed ISM (Industrial, Scientific and Medical) 2.4 G.Hz range and is rated at offering a data speed of 1.6 Mbps. The PC card transceivers communicate to an Access Point, which has a wired Ethernet connection to the network.

Future work

The present model is only a prototype for a future system. The connectivity issues are still being debated. To assure connectivity from the field to our intranet involves working with service providers. Dynamically assigned IP numbers will be of limited use because in that

scenario, the mobile clients will have to initiate all the communication. Networks like IRIDIUM [12], which consist of a system of low-level satellites, will be of interest to us. CDPD modems are becoming more common and they might become a viable option in the near future.

We would also be interested in using machines, which have more native Java support like the JavaStations [13]. Configurable systems running JavaOS [5] would allow faster execution of the MIDAS code and allow greater support for mobile agents.

Another possible design of MIDAS could involve the formation of an *Ad Hoc* network [14] consisting of the smart remote sensing devices communicating with the base-stations. This would make MIDAS more flexible and easy to configure. The installation time and the overhead involved in changing the configuration would be significantly reduced.

Conclusions

MIDAS is an experimental system, which has been built to study the impact of greater interaction to improve remote data acquisition. MIDAS has been designed in response to meet the requirements of researchers who require fine grain control over the instruments. It faces many of the problems inherent to mobile computers. It handles the command and control problem by transmitting a predicate, which must be true for the command to be executed. The focus has been on reliability and maintainability. We are of the opinion that such systems are feasible. We expect to deploy this system in the first half of 1998. Embedded systems with local intelligence will become more widespread with increasing system support.

References

[1] ARGOS User Manual version 1.0, January 1996.

[2] National Oceanic and Atmospheric Administration (NOAA)
http://www.noaa.gov

[3] Satyanarayanan, M.
Fundamental Challenges in Mobile Computing.
Proceedings of the fifteenth annual ACM symposium on Principles of Distributed Computing, May 23-26, 1996.

[4] Terry, D.B.
Caching Hints in Distributed Systems.
IEEE Transactions in Software Engineering, January, 1987.

[5] Peter, M., Susan, K., Douglas, K., Tom, S.
JAVAOS[tm]: A STANDALONE JAVA[tm] ENVIRONMENT
http://java.sun.com/products/javaos/javaos.white.html

[6] The JavaChip White Paper
http://www.sun.com/microelectronics/java/

[7] Microsoft WindowsCE
http://www.microsoft.com/windowsce

[8] William, B.M.
R&D Required for Remote Sensing
Sea Technology, Vol35, N11, 1994.

[9] Larry, T.C, Tatsuya, S.
Designing Mobile Computing Systems using Distributed Objects
IEEE Communications Magazine, February 1997.

[10] Lubomir, F.B., Michael, B.D., Munehiro, F.
Mobile Network Agents
Encyclopedia of Electrical and Electronics Engineering, John Wiley & Sons, Inc., 1998
(An introductory overview of mobile agents technologies).

[11] Wireless LAN Interoperability Forum
http://www.wlif.com/

[12] A New Generation of Satellite Communications
Iridium Today, Fall 1997.
http://www.iridium.com/public/public.html

[13] The JavaStation White Paper
http://www.sun.com/javastation/whitepapers/javastation/javast_ch1.html

[14] Johnson, D.B.
Routing in ad hoc networks of mobile hosts
Proceedings of the Workshop on Mobile Computing Systems and Applications
IEEE Computer Society Press, 1994